CSci 5299

Senior Thesis

Parallel Implementations of Neural Network Training: Two Back-Propagation Approaches

Jeffrey Dean
ID #1321294

Introduction

The back-propagation algorithm for training neural networks is used in a variety of applications due to its simplicity and ease of implementation. This paper assumes the reader is somewhat familar with neural networks in general and the back-propagation algorithm in particular. An introduction to these topics can be found in [Rume 86]. The simplicity and widespread use of the back-propagation algorithm make it a logical starting point for developing a parallel algorithm for training neural networks. The work presented in this paper focuses on two back-propagation based approaches to training a neural network in parallel.

The first approach is a pattern partitioning approach. In this approach, the entire network is represented on each processor and the set of input patterns is divided among the available processors. Each processor independently computes the weight changes necessary to minimize the error for its subset of the input patterns. After each processor has computed these weight changes, they are combined and broadcast to all processors.

The second approach involves partitioning the neurons of the neural network among the available processors and having each processor propagate the entire set of input patterns. The advantage of this approach is that changes do not need to be combined among the processors. However, communication is still needed during the propagation phase since each processor now represents only a portion of the entire network.

The pattern partitioning approach was implemented first due to its simplicity. Given n input patterns and p processors, each processor was given n/p input patterns (with appropriate rounding for non-integer values, of course) and the entire set of weights for the network. During each epoch, each processor propagates its input patterns through the network, and then back-propagates the error values. At some point, the processors need to combine the delta values resulting from the back-propagation phase.

Two different approaches to this combination were attempted. The first involved saving the results of the pattern propagation and the delta values from the back-propagation in matrices and then combining these matrices. This turned out to be a naive approach since the amount of communication that needed to be done increased linearly with the number of input patterns. A better approach was to collect the weight delta values for all of a processor's n/p patterns and then to combine these weight delta matrices by summing them. This approach provided much better scalability.

The network-partitioned approach that was implemented involves dividing the e network elements among the p processors. The model of a software pipeline is used, with

the processors communicating in a ring fashion. Patterns are placed in one end of the pipeline, and as they flow through the pipeline are processed by each processor's neurons. One processor does no actual processing of neurons but acts as a monitoring processor, placing messages into the pipeline, retrieving them from the end of the pipeline and keeping track of bookeeping tasks. The network elements are divided among the other p-1 processors.

Measures of Efficiency

The performance of a parallel algorithm can be split into two components: the time spent doing useful work, and the time spent idling or in communications. Call these two components t^i_{calc}, the time spent doing useful calculations, and t^i_{comm}, the time spent in communcations and not performing useful work (the superscript i denotes the processor number). The sum of t^i_{calc} for all processors is defined as T_{calc}, and the sum of t^i_{comm} for all processors is T_{comm}. The total time taken by an algorithm on p processors is:

$$T = \sum_{i=1}^{p} \left(t^i_{calc} + t^i_{comm}\right) = T_{calc} + T_{comm} = pT_p$$

where T_p is the execution time on p processors.

Problem Size W: The problem size is defined as the total amount of computation done by the best known sequential algorithm.

Speedup S: The speedup on p processors is defined as ratio of $\frac{T_1}{T_p}$, where T_1 is the time of the best sequential algorithm.

Efficiency E: Efficiency measures the effective utilization of p processors and is defined as $\frac{S_p}{p}$. Essentially measures what percentage of a linear speedup was obtained on p processors.

Iso-efficiency - The iso-efficiency function describes the relationship between W and p that must be maintained in order to keep a constant efficiency. See [Gupt 1990] for a complete discussion.

Definitions

In order to analyze the efficiency of these algorithms, we need some common definitions.

The inputs to a neural network consist of:

n input patterns, and

m neuron layers, each of size k, along with $(m - 1)$ weight matrices of size $k \times k$ connecting the layers. Note that we are assuming that each layer has the same number of neurons for the purposes of our analysis. The programs that were devloped allow differing number of neurons for each layer. This is generally the case in actual neural network applications.

We are given p processors.

From these values, we can derive some useful expressions:

M, the sizes of all weight matrices, is $(m - 1)k^2$.

C, the number of hidden and ouput layer neurons, is $(m - 1)k$.

Pattern-Partitioned Approach

The pattern partitioned approach on a hypercube is fairly simple to implement. The host processor send the weight matrices for the entire network to each processor, along with the subset of n/p input patterns that each processor will be handling. During an epoch, each processor calculates the weight delta values for its pattern set by propagating its inputs through the network. These weight delta matrices are then sent up a tree (mapped onto a hypercube in the usual manner). At each non-leaf node, the two sets of matrices are combined using matrix addition. The node at the root of the tree then has the weight delta matrices for the entire input pattern set, and broadcasts these to all the other processors (in log time, using the typical hypercube broadcasting method). Each processor then updates its weights and begins processing its input patterns for the next epoch.

The sequential algorithm takes time proportional to M for each input pattern. Thus, the sequential algorithm's time per epoch is $O(nM)$.

Analysis

The computation time per processor is:

$$t_{calc}^{i} = \frac{n}{p} M$$

The communication time per processor is:

$$t_{comm}^{i} = M \log p$$

Thus, the speedup of the pattern-partitioned approach is:

$$S = \frac{nM}{\frac{nM}{p} + M \log p}$$

The efficiency is:

$$E = \frac{nM}{nM + Mp \log p} = \frac{1}{1 + \frac{p \log p}{n}}$$

In order to maintain constant efficiency, $\frac{p \log p}{n}$ must remain constant. Thus, the iso-efficiency function is:

$$n \propto p \log p$$

Experimental Results

Several different sets of input data were generated in order to evaluate the performance of the pattern-partitioned approach. The intuitive result that better speedups would be obtained with larger numbers of input patterns held. Speedup curves for three of these input sets are depicted below:

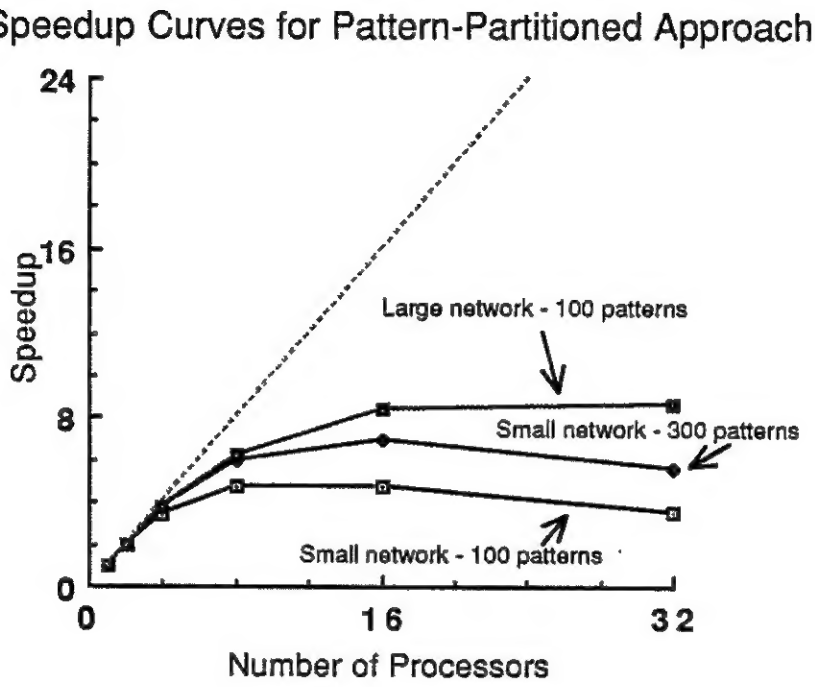

Fig. 1. Speedup curves for pattern-partitioned approach. The small network contained 3 layers with 9, 4 and 2 neurons. The large network contained 3 layers with 10, 21 and 10 neurons.

Pipelined Approach

The second method that was implemented involved a network-partitioning, rather than a pattern-partitioning approach approach. During an epoch, a given pattern will be handled by every processor in the system, but that processor will only do a portion of the processing for that pattern. Only hidden and output layer nodes are represented on processors; since the input layer has no weights incident on it, any processing of input layer nodes is assumed to be handled by the first hidden layer. One processor is assigned the role of manager and is responsible for filling the pipeline with messages, collecting the output at the other end of the pipeline, and various bookeeping tasks. The C hidden and output layer neurons are evenly divided among the remaining p - 1 processors[1]. Initially, the managing processor generates messages to completely fill the pipeline; thereafter, it receives output from the end of the pipeline and continues to ensure that the pipeline is full. For a complete discussion of the algorithm, see [Alle 89].

Each message in this approach contains the input pattern, target output pattern, a placeholder for the input to and output from the current layer, and a set of m vectors to hold the error information for each layer. The size of each message is O (mk).

Analysis

The computation time per processor per epoch is:

$$t_{calc}^{i} = \frac{n}{p} M = \frac{n}{p} mk^2$$

The communication time per processor epoch is:

$$t_{comm}^{i} = nC = nmk$$

Thus, the speedup of the pipelined network-partitioned approach is:

$$S = \frac{nmk^2}{\frac{nmk^2}{p} + nmk}$$

[1]Allen and Saha [ALLE 89] conducted experiments which showed that output layer neurons take 3 times as long to process as hidden layer neurons, so some load balancing may be necessary to offset this difference. However, this was not taken into account in this implementation.

The efficiency is:

$$E = \frac{nmk^2}{nmk^2 + pnmk} = \frac{1}{1 + \frac{p}{k}}$$

In order to maintain constant efficiency, $\frac{p}{k}$ must remain constant. Thus, the iso-efficiency function is:

$$k \propto p$$

Experimental Results

Several neural networks of various sizes were trained using this pipelined approach. Due to a limitation on message sizes between processors on the NCUBE multicomputer, results were only analyzed up to 32 processors. The calculations do not include the managing processor, since its functions could also be handled by the host processor. Thus, on a 3-dimensional hypercube, 7 processors were performing training, so the speedup is presented for 7 processors, not 8 processors. On the graph below, there are two curves: one is for the pipelined approach of a moderate sized network and the other is for the same inputs under the pattern-partitioned approach described above. Speedup curves for both methods are shown.

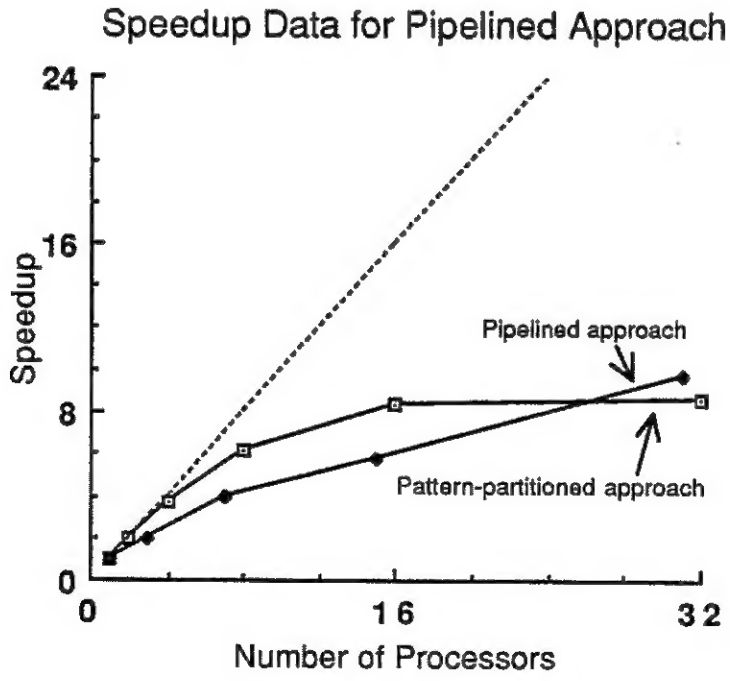

Fig 2. Data for large network dataset (3 layers, 10, 21 and 10 neurons per layer) for pipelined and pattern-partitioned approaches.

Discussion

Both approaches seemed to perform fairly well. The pattern-partitioned approach is less scalable than the pipelined approach, but performs better with a smaller number of processors. The pipelined approach could be improvemed in several ways. One way would be to balance the processing times taken for each processor in the pipeline by performing some analysis on the time taken for hidden and output layer neurons rather than simply dividing the available neurons evenly among the processors. This would eliminate any bottlenecks caused by differences in processor speed, and would increase concurrency. Another possibility is to have a two-way pipeline, in which messages flow to the end of the network, and then flow backwards to implement the back-propagation phase rather than saving the results of the previous epoch to be cycled through in the next epoch. This would reduce the message size by a factor of m and would undoubtedly increase the efficiency of the algorithm.

References

[Alle 89] Allen, Wayne and Avijit Saha, *Parallel Neural-Network Simulation Using Back-propagation for the ES-Kit Environment*, Microelectronics and Computer Technology Corporation, Austin, Texas, 1989.

[Ghos 89] Ghosh, Joydeep, and Kai Hwang, "*Mapping Neural Networks onto Message-Passing Multicomputers,*" Journal of Parallel and Distributed Computing, April 1989.

[Gupt 90] Gupta, Anshul and Vipin Kumar, *On the Scalability of FFT on Parallel Computers*, Dept. of Computer Science, University of Minnesota, 1990.

[Pome 88] Pomerleau, Dean A., George L. Gusciora, David S. Touretzky and H.T. Kung, "*Neural Network Simulation at Warp Speed: How We Got 17 Million Connections per Second,*" Internation Conference on Systolic Arrays, 1988.

[Rume 86] Rumelhart, D. E., and J.L. McClelland, editors, "*Parallel Distributed Processing: Explorations in the Microstructure of Cognition,*" Bradford Books/MIT Press, Cambridge, Mass., 1986.

```
/* net.h
*/
/* #define DEBUG */
/* #define QUICKF */
/* #define IBM */

#define ESC             27
#define ERRORLEVEL      0.02
#define ITEMS           8
#define MAXLAYERS       5
#define FALSE           0
#define TRUE            1
#define MAXDATA      15000
#define MAXBUFLEN   ((MAXDATA+4)*sizeof(float))
#define MAXCUBDIM       6
#define MS_PER_TICK     0.166
#define LBUFLEN       512
#define SMALLBUFLEN    10

/* message types for inter-processor communication */
#define M_INITIAL       1
#define M_PATTERN       2
#define M_WEIGHTS       3
#define M_UPDATE        4
#define M_OUTPUT        5
#define M_ERRORS        6
#define M_ERRORS2       7
#define M_TIME          8

/* typedefs and prototypes for dynamic storage of arrays */
typedef float *PFLOAT;
typedef PFLOAT VECTOR;
typedef PFLOAT *MATRIX;
typedef struct layerrec {
                int     lSize;
                int     wSize;
                VECTOR  del;
                MATRIX  wdel;
                MATRIX  w;
                VECTOR  o;
        } LAYER;
typedef struct vectormsg {
                int     v_id;
                int     v_cols;
                float   v_data[MAXDATA];
        } VECTORMSG, *VM_PTR;
typedef struct matrixmsg {
                int     m_id;
                int     m_rows;
                int     m_cols;
                float   m_data[MAXDATA];
        } MATRIXMSG, *MM_PTR;
typedef struct initialmsg {
                int     i_layers;
                int     i_iters;
                int     i_npats;
                int     i_startpat;
                int     i_endpat;
                long    i_seed;
                float   i_eta;
                float   i_alpha;
                float   i_epsilon;
                int     i_lsize[MAXLAYERS];
        } INITIALMSG, *IM_PTR;
typedef struct updatemsg {
                int     u_iter;
                int     u_maxiter;
                float   u_error;
        } UPDATEMSG, *UM_PTR;


#ifdef IBM
void  VectorAllocate(VECTOR *vector, int nCols);
void  AllocateCols(PFLOAT matrix[], int nRows, int nCols);
void  MatrixAllocate(MATRIX *pmatrix, int nRows, int nCols);
void  MatrixFree(MATRIX matrix,  int nRows);
void  LayerAllocate(LAYER *layer, int layersize, int wtsize);
void  LayerFree(LAYER *layer);
float f (float x);
#else
void  VectorAllocate();
void  AllocateCols();
void  MatrixAllocate();
void  MatrixFree();
void  LayerAllocate();
void  LayerFree();
float f ();
#endif
```

```
/* network.c

   Common routines to both the host and node programs.  Contains
   routines for allocating vectors and matrices, and for sending
   vectors and matrices as messages.
*/
/*      stdio.h

        Definitions for stream input/output.

        Copyright (c) Borland International 1987,1988
        All Rights Reserved.
*/
#define NULL      0
/*
#include <stdio.h>
#ifdef IBM
#include <stdlib.h>
#include <conio.h>
#endif
#include <math.h>
#include <ctype.h>
#include <string.h>
#include "net.h"

*/
/*--------------- Vector and Matrix message passing routines --------------*/
#ifdef IBM
int SendVector(VECTOR vector, int nCols, VM_PTR mbuf, int dest, int msg_id)
#else
int SendVector(vector, nCols, mbuf, dest, msg_id)
VECTOR vector;
int nCols;
VM_PTR mbuf;
int dest;
int msg_id;
#endif
{
   register int i;
   int buflen, flag;
   long result;

   mbuf->v_cols = nCols;
   mbuf->v_id   = msg_id;
#ifdef HOST
   swap (&mbuf->v_cols, sizeof (int));
   swap (&mbuf->v_id, sizeof (int));
#endif
   for (i = 0; i < nCols; i++)
   {
      mbuf->v_data[i] = vector [i];
#ifdef HOST
      swap (&(mbuf->v_data[i]), sizeof(float));
#endif
   }
   buflen = 2 * sizeof(int) + nCols * sizeof (float);

#ifdef HOST
   result = nwrite (chan, (char *)mbuf, buflen, dest, msg_id);
   return (result);
#else
   nwrite ((char *)mbuf, buflen, dest, msg_id, &flag);
   return (flag);
#endif
}

#ifdef IBM
int GetVector(VECTOR vector, int nCols, VM_PTR mbuf, int from, int msg_id)
#else
int GetVector(vector, nCols, mbuf, from, msg_id)
VECTOR vector;
int nCols;
VM_PTR mbuf;
int from;
int msg_id;
#endif
{
   register int i;
   int buflen, msgnode, type, flag;
   long result;

   buflen = 2 * sizeof(int) + nCols * sizeof (float);
   msgnode = from;
   type = msg_id;

#ifdef HOST
   result = nread (chan, (char *)mbuf, buflen, &msgnode, &type);
#else
   nread ((char *)mbuf, buflen, &msgnode, &type, &flag);
#endif
   for (i = 0; i < nCols; i++)
   {
#ifdef HOST
      swap (&(mbuf->v_data[i]), sizeof(float));
#endif
      vector [i] = mbuf->v_data[i];
   }
#ifdef HOST
   return (result);
#else
   return (flag);
#endif
}

#ifdef IBM
int SendMatrix(MATRIX matrix, int nRows, int nCols,
               MM_PTR mbuf,   int dest, int msg_id)
#else
int SendMatrix(matrix, nRows, nCols, mbuf, dest, msg_id)
MATRIX matrix;
int nRows;
int nCols;
MM_PTR mbuf;
int dest;
int msg_id;
#endif
{
   register int i, j;
   int buflen, flag;
   long result;

   mbuf->m_rows = nRows;
   mbuf->m_cols = nCols;
   mbuf->m_id   = msg_id;
#ifdef HOST
   swap (&mbuf->m_rows, sizeof (int));
   swap (&mbuf->m_cols, sizeof (int));
```

```
   swap (&mbuf->m_id, sizeof (int));
#endif
   for (i = 0; i < nRows; i++)
      for (j = 0; j < nCols; j++)
      {
         mbuf->m_data[i*nCols+j] = matrix [i][j];
#ifdef HOST
         swap (&(mbuf->m_data[i*nCols+j]), sizeof(float));
#endif
      }
   buflen = 3 * sizeof(int) + nRows * nCols * sizeof (float);

#ifdef HOST
   result = nwrite (chan, (char *)mbuf, buflen, dest, msg_id);
   return (1);
#else
   nwrite ((char *)mbuf, buflen, dest, msg_id, &flag);
   return (flag);
#endif
}

#ifdef IBM
int GetMatrix(MATRIX matrix, int nRows, int nCols,
              MM_PTR mbuf,   int from,  int msg_id)
#else
int GetMatrix(matrix, nRows, nCols, mbuf, from, msg_id)
MATRIX matrix;
int nRows;
int nCols;
MM_PTR mbuf;
int from;
int msg_id;
#endif
{
   register int i, j;
   int buflen, msgnode, type, flag;
   long result;

   buflen = 3 * sizeof(int) + nRows * nCols * sizeof (float);
   msgnode = from;
   type = msg_id;

#ifdef HOST
   result = nread (chan, (char *)mbuf, buflen, &msgnode, &type);
#else
   nread ((char *)mbuf, buflen, &msgnode, &type, &flag);
#endif
   for (i = 0; i < nRows; i++)
      for (j = 0; j < nCols; j++)
      {
#ifdef HOST
         swap (&(mbuf->m_data[i*nCols+j]), sizeof(float));
#endif
         matrix [i][j] = mbuf->m_data[i*nCols+j];
      }
#ifdef HOST
   return (result);
#else
   return (flag);
#endif
}

/*-------------------- Array storage allocation routines ------------------*/
/* Allocate space for vector of float cells for
   one dimensional dynamic vector[cols]
*/
#ifdef IBM
void VectorAllocate(VECTOR *vector, int nCols)
#else
void VectorAllocate(vector, nCols)
VECTOR *vector;
int nCols;
#endif
{
   if ((*vector = (VECTOR) calloc(nCols, sizeof(float))) == NULL)
   {
   }
}


/* Allocate space for columns (float cells) for
   dynamic two dimensional matrix[rows][cols]
*/
#ifdef IBM
void AllocateCols(PFLOAT matrix[], int nRows, int nCols)
#else
void AllocateCols(matrix, nRows, nCols)
PFLOAT matrix[];
int nRows;
int nCols;
#endif
{
   int  i;

   for (i = 0;  i < nRows;  i++)
      VectorAllocate(&matrix[i], nCols);
}

/* Allocate space for a two dimensional dynamic matrix [rows] [cols]
*/

#ifdef IBM
void MatrixAllocate(MATRIX *pmatrix, int nRows, int nCols)
#else
void MatrixAllocate(pmatrix, nRows, nCols)
MATRIX *pmatrix;
int nRows;
int nCols;
#endif
{
   if ( (*pmatrix  =  (MATRIX) calloc(nRows,  sizeof(PFLOAT) ) )   ==  NULL)
   {
   }

   AllocateCols(*pmatrix, nRows, nCols);
}

/* free space for two dimensional dynamic array */
#ifdef IBM
void MatrixFree(MATRIX matrix,  int nRows)
#else
void MatrixFree(matrix, nRows)
MATRIX matrix;
int nRows;
#endif
{
   int   i;
   for (i = 0;  i < nRows;  i++)
```

```
      free(matrix[i]);
   free(matrix);
}

#ifdef IBM
void  LayerAllocate(LAYER *layer, int layersize, int wtsize)
#else
void  LayerAllocate(layer, layersize, wtsize)
LAYER *layer;
int layersize;
int wtsize;
#endif
{
   layer->lSize = layersize;
   layer->wSize = wtsize;
/* w and wdel must contain room for the node bias */
   MatrixAllocate (&(layer->w), layersize, wtsize+1);
   MatrixAllocate (&(layer->wdel), layersize, wtsize+1);
   VectorAllocate (&(layer->o), layersize);
   VectorAllocate (&(layer->del), layersize);
}

#ifdef IBM
void  LayerFree(LAYER *layer)
#else
void  LayerFree(layer)
LAYER *layer;
#endif
{
   MatrixFree (layer->w, layer->lSize);
   MatrixFree (layer->wdel, layer->lSize);
   free (layer->o);
   free (layer->del);
}

#ifdef IBM
float f(float x)
#else
float f(x)
float x;
#endif
{
#ifdef QUICKF
   static int    doneinit = FALSE;
   static VECTOR v;
   static int    vsize    = 1000;
   static float  lo       = -5.0;
   static float  hi       =  5.0;
   static float  step;
   register int   i;
   float y, y2, d;

   if (!doneinit)
   {
      VectorAllocate (&v, vsize);
      step = (hi - lo) / vsize;
      for (i = 0, y = lo; i < vsize; i++, y+=step)
         v[i] = 1.0 / (1.0 + exp (-y));
      doneinit = TRUE;
   }
   if (x < lo || x >= hi)
      return (1.0 / (1.0 + exp (-x)));
   else
   {
      i = (x - lo) / step;
      y = (x - lo + i * step) / step;
      return (v[i] + y * (v [i+1] - v [i]));
   }

#else

   return (1.0 / (1.0 + exp (-x)));
#endif
}

power (cof, exp)
int   cof, exp;

{
int     ii, result;

result = 1;
for(ii = 0; ii < exp; ii++) {
        result = result * cof;
}
return(result);
}

swap(data,nbytes)  /* this funtion performs the byte reversal of a data
                      word of size = nbytes */
char data[];
int nbytes;
  {
  int left,right,temp;

  left = 0; right = nbytes - 1;
  while(right > left)
    {
    temp = data[left]; data[left++] = data[right]; data[right--] = temp;
    }
  }
```

```
.ethost.c

'effrey Dean                                    CSci 5299
 D 1321294                                      Term Project

lack-propagation neural network - Host program

'ine HOST

:lude <stdio.h>
ief IBM
:lude <stdlib.h>
:lude <conio.h>
iif
:lude <math.h>
:lude <ctype.h>
:lude <string.h>
:lude "net.h"

ief IBM
   ShowMatrix(MATRIX m, int nrows, int ncols);
se
   ShowMatrix();
iif

efine storage for net layers */
Arrays for inputs, outputs, deltas, weights & targets */
ER    net[MAXLAYERS];
      lSize[MAXLAYERS];
TOR   PatternID;
RIX   target;       /* target output */
RIX   input;        /* input patterns */

Cube communication variables */
      chan;
      cubedim, cubesize;
      type;
r     *mbuf, *lbuf, *nflags;
      b [SMALLBUFLEN];
at    bf[SMALLBUFLEN];
      mSize;
at    avgtime;

clude "network.c"

def IBM
d  main(int argc, char *argv[])
se
d  main(argc, argv)
 argc;
r *argv[];
dif

float eta    =  0.15,        /* default learning rate              */
      alpha  =  0.075;       /* default momentum factor            */
int   nReportErrors = 1;     /* error reporting frequency          */
float ErrorLevel = ERRORLEVEL; /* satisfactory error level         */
float error;                 /* latest sum squared error value     */
float pat_node;              /* patterns per node                  */
float pat_num;               /* current pattern number             */
int   i;           /* index variable                               */
int   j;
int   l,                     /* layer index                        */
      n,                     /* neuron index                       */
      p,                     /* index pattern number               */
      nPatterns,             /* number of patterns desired         */
      nLayers,               /* number of layers in network        */
      nIterations;           /* number of iterations desired       */
FILE  *fpInp,                /* run file                           */
      *fpResults,            /* resulting weights file             */
      *fpWeights;            /* initial weight file                */
char  szResults[66];         /* various filenames (pathnames)      */
char  szWeights[66];
char  *progname  =  *argv;   /* name of executable DOS 3.x only    */
char  c;

/* allocate message buffer */
mbuf = (char *)malloc(MAXBUFLEN);

printf ("Neural net simulation\n");

/* read optional - arguments */
for (; argc > 1;  argc--)
{
   char *arg = *++argv;

   if (*arg  !=  '-')
      break;

   switch (*++arg)
   {
      case 'e':   sscanf(++arg,  "%d",  &nReportErrors);   break;
      case 'd':   sscanf(++arg,  "%f",  &ErrorLevel);      break;
      default:    break;
   }
}

if (argc < 2)
{
   fprintf(stderr, "Usage:  %s (-en -df) runfilename\n",  progname);
   fprintf(stderr, "   -en   =>  report error every n iterations\n");
   fprintf(stderr, "   -df   =>  done if mean squared error < f\n");
   exit(1);
}

/* Open run file for reading */
if ((fpInp = fopen(*argv, "r"))   ==   NULL)
{
   fprintf(stderr, "%s: can't open file %s\n", progname, *argv);
   exit(1);
}

/* read and parse the run specification line; */
fscanf(fpInp,
    "%s %s %d %d %f %f %d",
    szResults,          /* output results file */
    szWeights,          /* initial weights file */
    &nPatterns,         /* number of patterns to learn */
    &nIterations,       /* number of iterations through the data */
    &eta,               /* learning rate */
    &alpha,             /* momentum factor */
    &nLayers);          /* number of layers in network */

/* get sizes of layers */
for (i = 0; i < nLayers; i++)
   fscanf(fpInp, "%d", &lSize [i]);

printf ("%d layers, %d patterns: ", nLayers, nPatterns);
```

```
for (i = 0; i < nLayers; i++)
   printf (" %d", lSize [i]);
printf ("\n");


/*----------allocate dynamic storage for all data ---------------*/
for (i = 0; i < nLayers; i++)
   LayerAllocate(&net [i], lSize [i], (i==0) ? 0:lSize [i-1]);

MatrixAllocate(&target,    nPatterns,    lSize [nLayers-1]);
MatrixAllocate(&input,     nPatterns,    lSize [0]);
VectorAllocate(&PatternID, nPatterns);

/*---------- Read the initial weight matrices: --------------------*/
if ((fpWeights = fopen(szWeights,"r"))  ==  NULL)
{
   fprintf(stderr,  "%s: can't open file %s\n",  progname, szWeights);
   exit(1);
}

/* read weights */
for (l = 1; l < nLayers; l++)
   for (n = 0; n < net [l].lSize; n++)
      for (i = 0; i <= net [l-1].lSize; i++)
      {
         fscanf(fpWeights,  "%f",      &(net [l].w[n][i]));
         net [l].wdel[n][i] = 0.0;
      }

fclose(fpWeights);

/*------------ Read in all patterns to be learned ------------------*/
for (p = 0;  p < nPatterns;  p++)
{
   for (i = 0;   i < net [0].lSize;   i++)
      if (fscanf(fpInp,  "%f",   &input[p][i])  != 1)
         goto  ALLPATTERNSREAD;

   /* read in target outputs for input patterns read */
   for (i = 0;  i < net [nLayers-1].lSize;  i++)
      fscanf(fpInp,  "%f",   &target[p][i]);

   fscanf(fpInp, "%f", &PatternID[p]);
}

LPATTERNSREAD:

if (p < nPatterns)
{
   fprintf(stderr, "%s:  %d out of %d patterns read\n",
           progname,  p,  nPatterns);
   nPatterns = p;
}

cubedim = MAXCUBDIM + 1;
while ((cubedim < 0) || (cubedim > MAXCUBDIM)) {
       printf(" Enter Cube Dimension (0 - %d)  : ", MAXCUBDIM);
       scanf("%d", &cubedim);
}

cubesize = power(2, cubedim);   /* number of processors */
lbuf = (char *)malloc(LBUFLEN);
printf ("Allocating hypercube.\nTry ");
i = 1;
do {
   printf ("%d..", i);
   i++;
} while ((chan = nopen(cubedim)) < 0);

/* load the node program */
printf ("\nLoading node program, chan = %d.\n", chan);
nloadm(chan, "node_x", -cubesize, nflags, lbuf, LBUFLEN);
free(lbuf);
printf ("Node program loaded to %d processors.\n", cubesize);

fprintf(stderr, "Setting up initial message to nodes.\n");
((IM_PTR)mbuf)->i_layers    = nLayers;
((IM_PTR)mbuf)->i_iters     = nIterations;
((IM_PTR)mbuf)->i_npats     = nPatterns;
((IM_PTR)mbuf)->i_seed      = 1L;
((IM_PTR)mbuf)->i_eta       = eta;
((IM_PTR)mbuf)->i_alpha     = alpha;
((IM_PTR)mbuf)->i_epsilon   = ErrorLevel;
printf ("Set up initial message.  Now swapping.\n");
swap (&((IM_PTR)mbuf)->i_layers, sizeof(int));
swap (&((IM_PTR)mbuf)->i_iters, sizeof(int));
swap (&((IM_PTR)mbuf)->i_npats, sizeof(int));
swap (&((IM_PTR)mbuf)->i_seed, sizeof(long));
swap (&((IM_PTR)mbuf)->i_eta, sizeof(float));
swap (&((IM_PTR)mbuf)->i_alpha, sizeof(float));
swap (&((IM_PTR)mbuf)->i_epsilon, sizeof(float));

for (i = 0; i < MAXLAYERS; i++)
{
   ((IM_PTR)mbuf)->i_lsize [i]  = lSize [i];
   swap (&((IM_PTR)mbuf)->i_lsize[i], sizeof(int));
}

pat_node = (float)nPatterns / cubesize;
pat_num = 0;
printf ("Now sending initial message to nodes.  Total patterns = %d\n",
        nPatterns);

for (i = 0; i < cubesize; i++)
{
   /* Synchronization step */
   type = M_INITIAL;
   nread (chan, (char *)b, sizeof (int), &i, &type);

   ((IM_PTR)mbuf)->i_startpat  = (int) pat_num;
   ((IM_PTR)mbuf)->i_endpat    = (int) (pat_num + pat_node - 1);
   printf ("node %d gets patterns %3d to %3d.\n", i,
   ((IM_PTR)mbuf)->i_startpat,
   ((IM_PTR)mbuf)->i_endpat);
   swap (&((IM_PTR)mbuf)->i_startpat, sizeof(int));
   swap (&((IM_PTR)mbuf)->i_endpat, sizeof(int));
   pat_num += pat_node;

   type = M_INITIAL;
   nwrite (chan, (char *)mbuf, sizeof(INITIALMSG), i, type);
}
printf ("Initial message sent.  Sending input matrix..\n");
printf ("Node ");
for (i = 0; i < cubesize; i++)
{
   printf ("%d...", i);
   SendMatrix (input,  nPatterns, lSize [0], (MM_PTR)mbuf,
               i, M_PATTERN);
```

```
      SendMatrix (target, nPatterns, lSize [nLayers-1], (MM_PTR)mbuf,
                  i, M_OUTPUT);
      for (j = 1; j < nLayers; j++)
         SendMatrix (net [j].w, lSize [j], lSize [j-1]+1, (MM_PTR)mbuf,
                     i, M_WEIGHTS);
   }
   printf ("\n");
   fprintf(stderr,  nIterations > 1  ?  "Training...\n"  :  "Testing\n");

   /* Now we await the results */
   do
   {
      i = -1;
      type = M_UPDATE;
      nread (chan, (char *)bf, 4 * sizeof (float), &i, &type);
      for (j = 0; j < 4; j++)
         swap (bf+j, sizeof(float));
      printf ("Node %d: Iteration %5.0f/%-5.0f  Error: %f, type = %5.0f\n",
                i, bf[0], bf[1], bf[2], bf [3]);
   } while (bf [3] != 1);

   /* Now get the weight matrices from node 0 */
   for (l = 1; l < nLayers; l++)
      GetMatrix (net [l].w, lSize [l], lSize [l-1]+1, (MM_PTR)mbuf,
                 0, M_WEIGHTS);

   /* Now get the time taken from each node */
   for (i = 0, avgtime = 0.0; i < cubesize; i++)
   {
      type = M_TIME;
      nread (chan, (char *)b, sizeof(int), &i, &type);
      swap (b, sizeof(int));
      printf ("Node %d took %f ms (%d x %f)\n", i, b[0]*MS_PER_TICK,
              b[0], MS_PER_TICK);
      avgtime += b[0] * MS_PER_TICK;
   }
   printf ("\nAverage time per node: %6.1f ms\n", avgtime / cubesize);

   printf ("Program terminating.\n");

   /*---------------- free dynamic storage for data ---------------------*/
   for (l = 0; l < nLayers; l++)
      LayerFree (&net [l]);
   MatrixFree(target,    nPatterns);
   MatrixFree(input,     nPatterns);
   free (PatternID);

   fclose(fpInp);                    /* close run file */
}

#ifdef IBM
int   ShowMatrix(MATRIX m, int nrows, int ncols)
#else
int   ShowMatrix(m, nrows, ncols)
MATRIX m;
int nrows;
int ncols;
#endif
{
   int i, j;

   for (i = 0; i < nrows; i++)
   {
      for (j = 0; j < ncols; j++)
         printf ("%5.4f%c", m[i][j], (j==ncols-1) ? '\n' : ' ');
#ifdef IBM
      if (i % 25 == 24) getch();
#endif
   }
   return (1);
}
```

```
/* netnode.c

   Jeffrey Dean                                    CSci 5299
   ID 1321294                                      Term Project

   Back-propagation neural network
*/
#define NODE

#ifdef IBM
#include <stdlib.h>
#endif
#include <math.h>
#include <ctype.h>
#include <string.h>
#include "net.h"

/* define storage for net layers */
/* Arrays for inputs, outputs, deltas, weights & targets */
LAYER    net[MAXLAYERS];
int      lSize[MAXLAYERS];
VECTOR   PatternID;
MATRIX   target;      /* target output */
MATRIX   input;       /* input patterns */
MATRIX   tempwdel;

long     ranseed = 1235734L;

/* Cube communication variables */
int      nodeno, proc, host, cubedim, flag;
int      type;
char     *mbuf;
int      b [SMALLBUFLEN];
float    bf[SMALLBUFLEN];

#include "network.c"

void  main()
{
   float eta   =  0.15,        /* default learning rate            */
         alpha =  0.075;       /* default momentum factor          */
   int   nReportErrors = 1;    /* error reporting frequency        */
   float ErrorLevel = ERRORLEVEL; /* satisfactory error level      */
   char  MonitorError = 0;     /* true when monitor error display  */
   float error;                /* latest sum squared error value   */
   int   i,                    /* index variables                  */
         j;
   int   mask;                 /* mask variable for broadcasting   */
   int   stepstowait;          /* steps to wait during broadcast   */
   int   msgnode;              /* node to send to during broadcast */
   int   l,                    /* layer index                      */
         n,                    /* neuron index                     */
         p,                    /* index pattern number             */
         q,                    /* index iteration number           */
         nPatterns,            /* number of patterns desired       */
         startPat,             /* starting pattern number for node */
         endPat,               /* ending pattern number for node   */
         nInputNodes,          /* number of input nodes            */
         nHiddenNodes,         /* number of hidden nodes           */
         nOutputNodes,         /* number of output nodes           */
         nLayers,              /* number of layers in network      */
         maxLSize,             /* maximum layer size               */
         nIterations;          /* number of iterations desired     */
   int   sTime;                /* Starting time                    */

   whoami (&nodeno, &proc, &host, &cubedim);

   /* allocate message buffer */
   mbuf = (char *)malloc(MAXBUFLEN);

   b[0] = 1;
   type = M_INITIAL;
   nwrite ((char *)b, sizeof (int), host, type, &flag);

   type = M_INITIAL;
   nread ((char *)mbuf, sizeof (INITIALMSG), &host, &type, &flag);

   nLayers     = ((IM_PTR)mbuf)->i_layers;
   nIterations = ((IM_PTR)mbuf)->i_iters;
   nPatterns   = ((IM_PTR)mbuf)->i_npats;
   startPat    = ((IM_PTR)mbuf)->i_startpat;
   endPat      = ((IM_PTR)mbuf)->i_endpat;
   ranseed     = ((IM_PTR)mbuf)->i_seed;
   eta         = ((IM_PTR)mbuf)->i_eta;
   alpha       = ((IM_PTR)mbuf)->i_alpha;
   ErrorLevel  = ((IM_PTR)mbuf)->i_epsilon;

   maxLSize = 0;
   for (i = 0; i < MAXLAYERS; i++)
   {
      lSize [i] = ((IM_PTR)mbuf)->i_lsize [i];
      maxLSize = ((lSize [i] > maxLSize) ? lSize [i] : maxLSize);
   }

   /*----------allocate dynamic storage for all data ---------------*/
   for (l = 0; l < nLayers; l++)
      LayerAllocate(&net [l], lSize [l], (l==0) ? 0:lSize [l-1]);

   MatrixAllocate(&target,    nPatterns,    lSize [nLayers-1]);
   MatrixAllocate(&input,     nPatterns,    lSize [0]);
   MatrixAllocate(&tempwdel,  maxLSize,     maxLSize+1);
   VectorAllocate(&PatternID, nPatterns);

   GetMatrix (input, nPatterns, lSize [0], (MM_PTR)mbuf, host,
              M_PATTERN);
   GetMatrix (target, nPatterns, lSize [nLayers-1], (MM_PTR)mbuf, host,
              M_OUTPUT);

   for (l = 1; l < nLayers; l++)
   {
      GetMatrix (net [l].w, lSize [l], lSize [l-1] + 1, (MM_PTR)mbuf, host,
                 M_WEIGHTS);
      /* initialize weight deltas to 0 */
      for (n = 0; n < lSize [l]; n++)
         for (j = 0; j <= lSize [l-1]; j++)
            net[l].wdel[n][j] = 0;
   }

   /* We now have all the matrices we need; start doing useful work */
   sTime = ntime();  /* get start time */

   /*---------------------- begin iteration loop -------------------------*/
   for (q = 0;  q < nIterations;  q++)
   {
      /* initialize weight deltas to momentum factor */
      for (l = 1; l < nLayers; l++)
         for (n = 0; n < lSize [l]; n++)
            for (j = 0; j <= lSize [l-1]; j++)
```

```
            net [l].wdel [n][j] = alpha * net[l].wdel[n][j];

   /* initialize error level to 0 */
   error = 0.0;

   for (p = startPat;  p <= endPat;  p++)
   {
      /*--------- copy current pattern to outputs for level 0 --------*/
      for (n = 0; n < lSize [0]; n++)
         net[0].o[n] = input[p][n];

      /*------------ evaluate network using current weights ---------*/
      /* Sum outputs from layer to layer over all possible edges */
      for (l = 1; l < nLayers; l++)
      {
         for (n = 0; n < net[l].lSize; n++)
         {
             float sum = net[l].w[n][net [l-1].lSize];

             for (i = 0; i < net [l-1].lSize; i++)
                sum += net[l].w[n][i] * net[l-1].o[i];

             net[l].o [n] = f (sum);
         }
      }

      /* accumulate error term */
      for (n = 0; n < lSize [nLayers-1]; n++)
      {
         float temp = target [p][n] - net [nLayers-1].o [n];

         error += temp * temp;
      }

      /*-------------------- calculate output deltas -------------------*/
      /* Compute deltas for each output unit for the given pattern */
      for (n = 0; n < net [nLayers-1].lSize; n++)
      {
         float opn = net [nLayers-1].o [n];

         net [nLayers-1].del[n] =
            (target[p][n] - opn) * opn * (1.0 - opn);
      }

      /*-------------------- calculate hidden deltas -------------------*/
      for (l = nLayers-2; l > 0; l--)
         for (n = 0; n < net [l].lSize; n++)
         {
            float sum = 0.0;
            float opn = net [l].o [n];

            for (i = 0; i < net [l+1].lSize; i++)
               sum += net [l+1].del [i] * net [l+1].w [i][n];

            net [l].del [n] = sum * opn * (1.0 - opn);
         }

   /*---------------------- adapt weights ---------------------*/
   for (l = nLayers - 1; l > 0; l--)
   {
      for (n = 0; n < net [l].lSize; n++)
      {
         float dw;                    /* delta weight */

            /* calculate dw with added bias: */
            dw = eta * net [l].del [n];
            net [l].wdel [n][net [l].lSize] += dw;

            /* calculate new weights */
            for (i = 0; i < net [l-1].lSize; i++)
            {
               dw = eta * net[l].del [n] * net [l-1].o [i];
               net [l].wdel [n][i] += dw;
            }
         }
      }
      } /* end of pattern loop */

      /*************************************************************/
      /*                    communicate deltas                     */
      /*************************************************************/
      /* first go up a binary tree (mapped onto the cube), sending */
      /* the partial delta results for each nodes patterns.        */
      /*************************************************************/
      for (i = cubedim-1; i >= 0; i--)
      {
         mask = (1 << i);
         if (nodeno >= (mask << 1)) goto loopexit;
         msgnode = nodeno ^ mask;
         if ((nodeno & mask) == 0)
         {
            /* accumulate error values from lower level nodes */
            type = M_ERRORS;
            nread ((char *)bf, sizeof (float), &msgnode, &type, &flag);
            error += bf [0];

            /* Get weight delta matrices for each layer and add to
               current weight deltas */
            for (l = 1; l < nLayers; l++)
            {
               GetMatrix (tempwdel, lSize [l], lSize [l-1]+1,
                          (MM_PTR)mbuf, nodeno ^ mask, M_WEIGHTS);
               for (n = 0; n < lSize [l]; n++)
                  for (j = 0; j <= lSize [l-1]; j++)
                     net [l].wdel [n][j] += tempwdel [n][j];
            }
         }
         else
         {
            /* accumulate error values from lower level nodes */
            type = M_ERRORS;
            bf [0] = error;
            nwrite ((char *)bf, sizeof (float), msgnode, type, &flag);
            /* Send weight delta matrices for each layer */
            for (l = 1; l < nLayers; l++)
              SendMatrix (net [l].wdel, lSize [l], lSize [l-1]+1,
                          (MM_PTR)mbuf, nodeno ^ mask, M_WEIGHTS);
         }
      }

loopexit:

      /*************************************************************/
      /* Now broadcast the complete weight delta matrices to all   */
      /* nodes, starting with node 0, the root of the tree we used */
      /* above.                                                    */
      /*************************************************************/
      stepstowait = maxdiff (nodeno, 0);  /* returns -1 if equal */
```

```
   for(i = 0; i < cubedim; i++)
   {
      mask = (1 << i);
      msgnode = nodeno ^ mask;
      if (i > stepstowait)
      {
         /* accumulate error values from lower level nodes */
         type = M_ERRORS2;
         bf [0] = error;
         nwrite ((char *)bf, sizeof (float), msgnode, type, &flag);

         for (l = 1; l < nLayers; l++)
            SendMatrix (net [l].wdel, lSize[l], lSize [l-1]+1,
                        (MM_PTR)mbuf, nodeno ^ mask, M_WEIGHTS);
      }
      else if (i == stepstowait)
      {
         /* accumulate error values from lower level nodes */
         type = M_ERRORS2;
         nread ((char *)bf, sizeof (float), &msgnode, &type, &flag);
         error = bf [0];

         for (l = 1; l < nLayers; l++)
            GetMatrix (net [l].wdel, lSize[l], lSize [l-1]+1,
                       (MM_PTR)mbuf, nodeno ^ mask, M_WEIGHTS);
      }
   }

   /* Now adjust weights using entire wdel matrices we just
      received from the broadcast */
   for (l = 1; l < nLayers; l++)
      for (n = 0; n < lSize [l]; n++)
         for (j = 0; j <= lSize [l-1]; j++)
            net [l].w [n][j] += net [l].wdel [n][j];

   /*-------------------- Sum Squared Error -------------------------*/
   if (MonitorError  ||  (q % nReportErrors   ==   0))
   {
      /* Average error over all patterns */
      error  /=  (nPatterns * net [nLayers-1].lSize);

      if (nodeno == 0)
      {
         bf [0] = (float)q;
         bf [1] = (float)nIterations;
         bf [2] = error;
         bf [3] = 0;
         nwrite ((char *)bf, 4*sizeof(float), host, M_UPDATE, &flag);
      }

      MonitorError = 0;

      /* Terminate when error satisfactory */
      if (error < ErrorLevel)
         break;
   }
}
/* ---------------- end of iteration loop ------------------------------*/

if (nodeno == 0)
{
   bf [0] = (float)q;
   bf [1] = (float)nIterations;
   bf [2] = error;
   bf [3] = 1;
   nwrite ((char *)bf, 4*sizeof(float), host, M_UPDATE, &flag);

   for (l = 1; l < nLayers; l++)
      SendMatrix (net [l].w, lSize [l], lSize [l-1] + 1,
                  (MM_PTR)mbuf, host, M_WEIGHTS);
}

/* Send time taken back to the host */
b [0] = ntime() - sTime;
nwrite ((char *)b, sizeof(int), host, M_TIME, &flag);

endpcs();
}

int maxdiff (n1, n2)
int n1, n2;
{
   int i, j, mask;

   for (i = 0, j = 0; i < 8 * sizeof (int); i++)
   {
      mask = (1 << i);
      if ((n1 & mask) != (n2 & mask))
         j = i;
   }
   if (n1 == n2)
      return (-1);
   else
      return (j);
}
```

```
/* pipe.h
*/
/* #define DEBUG */
/* #define QUICKF */
/* #define IBM */

#define ESC             27
#define ERRORLEVEL      0.02
#define ITEMS           8
#define MAXLAYERS       5
#define MAXNEURONS      10  /* Maximum number of neurons per node */
#define FALSE           0
#define TRUE            1
#define MAXDATA     12000
#define MAXBUFLEN   ((MAXDATA+4)*sizeof(float))
#define MAXCUBDIM       6
#define MAX_PAT_MSG    16
#define MS_PER_TICK     0.166
#define LBUFLEN       512
#define SMALLBUFLEN    10
#define TEL_SIZE      300

/* message types for inter-processor communication */
#define M_INITIAL       1
#define M_PATTERN       2
#define M_WEIGHTS       3
#define M_UPDATE        4
#define M_OUTPUT        5
#define M_ERRORS        6
#define M_ERRORS2       7
#define M_TIME          8
#define M_WUPDATE       9
#define M_SYNCH         10
#define M_TELEGRAM      100

/* typedefs and prototypes for dynamic storage of arrays */
typedef unsigned int UINT;
typedef float *PFLOAT;
typedef PFLOAT VECTOR;
typedef PFLOAT *MATRIX;
typedef struct neuron {
                int     lNum;
                int     nNum;
                VECTOR  w;
                VECTOR  wdel;
        } NEURON, *NEURON_PTR;
typedef struct layerrec {
                int     lSize;
                int     wSize;
                VECTOR  del;
                MATRIX  wdel;
                MATRIX  w;
                VECTOR  o;
        } LAYER;
typedef struct patmsg {
                int     p_npats;
                int     p_snum;
                float   p_data[MAXDATA];
        } PATMSG, *PAT_PTR;
typedef struct vectormsg {
                int     v_id;
                int     v_cols;
                float   v_data[MAXDATA];
        } VECTORMSG, *VM_PTR;
typedef struct matrixmsg {
                int     m_id;
                int     m_rows;
                int     m_cols;
                float   m_data[MAXDATA];
        } MATRIXMSG, *MM_PTR;
typedef struct initialmsg {
                int     i_layers;
                int     i_iters;
                int     i_pats;
                int     i_startN;
                int     i_endN;
                int     i_Lnbr;
                int     i_Rnbr;
                long    i_seed;
                float   i_eta;
                float   i_alpha;
                float   i_epsilon;
                int     i_lsize[MAXLAYERS];
        } INITIALMSG, *IM_PTR;
typedef struct updatemsg {
                int     u_iter;
                int     u_maxiter;
                float   u_error;
        } UPDATEMSG, *UM_PTR;


#ifdef IBM
void  VectorAllocate(VECTOR *vector, int nCols);
void  AllocateCols(PFLOAT matrix[], int nRows, int nCols);
void  MatrixAllocate(MATRIX *pmatrix, int nRows, int nCols);
void  MatrixFree(MATRIX matrix,  int nRows);
void  LayerFree(LAYER *layer);
float f (float x);
UINT bintogray(UINT n);
UINT graytobin(UINT g);
void LandN (int lSize[], int num, int *L, int *N)
#else
void  VectorAllocate();
void  AllocateCols();
void  MatrixAllocate();
void  MatrixFree();
void  LayerFree();
float f ();
UINT bintogray();
UINT graytobin();
void LandN();
#endif
#ifdef NODE
int dprintf();

char dbuf [TEL_SIZE];  /* buffer for debugging messages */
#endif
```

```
#define PRINTCOND (nodeno <= 3)
/* pipework.c

   Common routines to both the host and node programs.  Contains
   routines for allocating vectors and matrices, and for sending
   vectors and matrices as messages.
*/
/*      stdio.h

        Definitions for stream input/output.

        Copyright (c) Borland International 1987,1988
        All Rights Reserved.
*/
#define NULL      0
/*
#include <stdio.h>
#ifdef IBM
#include <stdlib.h>
#include <conio.h>
#endif
#include <math.h>
#include <ctype.h>
#include <string.h>
#include "pipe.h"

*/
/*--------------- Vector and Matrix message passing routines --------------*/
#ifdef IBM
int SendVector(VECTOR vector, int nCols, VM_PTR mbuf, int dest, int msg_id)
#else
int SendVector(vector, nCols, mbuf, dest, msg_id)
VECTOR vector;
int nCols;
VM_PTR mbuf;
int dest;
int msg_id;
#endif
{
   register int i;
   int buflen, flag;
   long result;

   mbuf->v_cols = nCols;
   mbuf->v_id   = msg_id;
#ifdef HOST
   swap (&mbuf->v_cols, sizeof (int));
   swap (&mbuf->v_id, sizeof (int));
#endif
   for (i = 0; i < nCols; i++)
   {
      mbuf->v_data[i] = vector [i];
#ifdef HOST
      swap (&(mbuf->v_data[i]), sizeof(float));
#endif
   }
   buflen = 2 * sizeof(int) + nCols * sizeof (float);

#ifdef HOST
   result = nwrite (chan, (char *)mbuf, buflen, dest, msg_id);
   return (result);
#else
   nwrite ((char *)mbuf, buflen, dest, msg_id, &flag);
   return (flag);
#endif
}

#ifdef IBM
int GetVector(VECTOR vector, int nCols, VM_PTR mbuf, int from, int msg_id)
#else
int GetVector(vector, nCols, mbuf, from, msg_id)
VECTOR vector;
int nCols;
VM_PTR mbuf;
int from;
int msg_id;
#endif
{
   register int i;
   int buflen, msgnode, type, flag;
   long result;

   buflen = 2 * sizeof(int) + nCols * sizeof (float);
   msgnode = from;
   type = msg_id;

#ifdef HOST
   result = nread (chan, (char *)mbuf, buflen, &msgnode, &type);
#else
   nread ((char *)mbuf, buflen, &msgnode, &type, &flag);
#endif
   for (i = 0; i < nCols; i++)
   {
#ifdef HOST
      swap (&(mbuf->v_data[i]), sizeof(float));
#endif
      vector [i] = mbuf->v_data[i];
   }
#ifdef HOST
   return (result);
#else
   return (flag);
#endif
}

#ifdef IBM
int SendMatrix(MATRIX matrix, int nRows, int nCols,
               MM_PTR mbuf,   int dest, int msg_id)
#else
int SendMatrix(matrix, nRows, nCols, mbuf, dest, msg_id)
MATRIX matrix;
int nRows;
int nCols;
MM_PTR mbuf;
int dest;
int msg_id;
#endif
{
   register int i, j;
   int buflen, flag;
   long result;

   mbuf->m_rows = nRows;
   mbuf->m_cols = nCols;
   mbuf->m_id   = msg_id;
#ifdef HOST
   swap (&mbuf->m_rows, sizeof (int));
   swap (&mbuf->m_cols, sizeof (int));
```

```
   swap (&mbuf->m_id, sizeof (int));
#endif
   for (i = 0; i < nRows; i++)
      for (j = 0; j < nCols; j++)
      {
         mbuf->m_data[i*nCols+j] = matrix [i][j];
#ifdef HOST
         swap (&(mbuf->m_data[i*nCols+j]), sizeof(float));
#endif
      }
   buflen = 3 * sizeof(int) + nRows * nCols * sizeof (float);

#ifdef HOST
   result = nwrite (chan, (char *)mbuf, buflen, dest, msg_id);
   return (1);
#else
   nwrite ((char *)mbuf, buflen, dest, msg_id, &flag);
   return (flag);
#endif
}

#ifdef IBM
int GetMatrix(MATRIX matrix, int nRows, int nCols,
              MM_PTR mbuf,   int from,  int msg_id)
#else
int GetMatrix(matrix, nRows, nCols, mbuf, from, msg_id)
MATRIX matrix;
int nRows;
int nCols;
MM_PTR mbuf;
int from;
int msg_id;
#endif
{
   register int i, j;
   int buflen, msgnode, type, flag;
   long result;

   buflen = 3 * sizeof(int) + nRows * nCols * sizeof (float);
   msgnode = from;
   type = msg_id;

#ifdef HOST
   result = nread (chan, (char *)mbuf, buflen, &msgnode, &type);
#else
   nread ((char *)mbuf, buflen, &msgnode, &type, &flag);
#endif
   for (i = 0; i < nRows; i++)
      for (j = 0; j < nCols; j++)
      {
#ifdef HOST
         swap (&(mbuf->m_data[i*nCols+j]), sizeof(float));
#endif
         matrix [i][j] = mbuf->m_data[i*nCols+j];
      }
#ifdef HOST
   return (result);
#else
   return (flag);
#endif
}

/*-------------------- Array storage allocation routines ------------------*/
/* Allocate space for vector of float cells for
   one dimensional dynamic vector[cols]
*/
#ifdef IBM
void VectorAllocate(VECTOR *vector, int nCols)
#else
void VectorAllocate(vector, nCols)
VECTOR *vector;
int nCols;
#endif
{
   if ((*vector = (VECTOR) calloc(nCols, sizeof(float))) == NULL)
   {
   }
}


/* Allocate space for columns (float cells) for
   dynamic two dimensional matrix[rows][cols]
*/
#ifdef IBM
void AllocateCols(PFLOAT matrix[], int nRows, int nCols)
#else
void AllocateCols(matrix, nRows, nCols)
PFLOAT matrix[];
int nRows;
int nCols;
#endif
{
   int  i;

   for (i = 0;  i < nRows;  i++)
      VectorAllocate(&matrix[i], nCols);
}

/* Allocate space for a two dimensional dynamic matrix [rows] [cols]
*/

#ifdef IBM
void MatrixAllocate(MATRIX *pmatrix, int nRows, int nCols)
#else
void MatrixAllocate(pmatrix, nRows, nCols)
MATRIX *pmatrix;
int nRows;
int nCols;
#endif
{
   if ( (*pmatrix  =  (MATRIX) calloc(nRows,  sizeof(PFLOAT) ) )   ==  NULL)
   {
   }

   AllocateCols(*pmatrix, nRows, nCols);
}

/* free space for two dimensional dynamic array */
#ifdef IBM
void MatrixFree(MATRIX matrix,  int nRows)
#else
void MatrixFree(matrix, nRows)
MATRIX matrix;
int nRows;
#endif
{
   int   i;
   for (i = 0;  i < nRows;  i++)
```

```
      free(matrix[i]);
   free(matrix);
}

#ifdef IBM
void  LayerAllocate(LAYER *layer, int layersize, int wtsize)
#else
void  LayerAllocate(layer, layersize, wtsize)
LAYER *layer;
int layersize;
int wtsize;
#endif
{
   layer->lSize = layersize;
   layer->wSize = wtsize;
/* w and wdel must contain room for the node bias */
   MatrixAllocate (&(layer->w), layersize, wtsize+1);
   MatrixAllocate (&(layer->wdel), layersize, wtsize+1);
   VectorAllocate (&(layer->o), layersize);
   VectorAllocate (&(layer->del), layersize);
}

#ifdef IBM
void  LayerFree(LAYER *layer)
#else
void  LayerFree(layer)
LAYER *layer;
#endif
{
   MatrixFree (layer->w, layer->lSize);
   MatrixFree (layer->wdel, layer->lSize);
   free (layer->o);
   free (layer->del);
}

#ifdef IBM
float f(float x)
#else
float f(x)
float x;
#endif
{
#ifdef QUICKF
   static int    doneinit = FALSE;
   static VECTOR v;
   static int    vsize    = 1000;
   static float  lo       = -5.0;
   static float  hi       =  5.0;
   static float  step;
   register int   i;
   float y, y2, d;

   if (!doneinit)
   {
      VectorAllocate (&v, vsize);
      step = (hi - lo) / vsize;
      for (i = 0, y = lo; i < vsize; i++, y+=step)
         v[i] = 1.0 / (1.0 + exp (-y));
      doneinit = TRUE;
   }
   if (x < lo || x >= hi)
      return (1.0 / (1.0 + exp (-x)));
   else
   {
      i = (x - lo) / step;
      y = (x - lo + i * step) / step;
      return (v[i] + y * (v [i+1] - v [i]));
   }

#else

   return (1.0 / (1.0 + exp (-x)));
#endif
}

power (cof, exp)
int   cof, exp;

{
int     ii, result;

result = 1;
for(ii = 0; ii < exp; ii++) {
        result = result * cof;
}
return(result);
}

swap(data,nbytes)  /* this funtion performs the byte reversal of a data
                      word of size = nbytes */
char data[];
int nbytes;
  {
  int left,right,temp;

  left = 0; right = nbytes - 1;
  while(right > left)
    {
    temp = data[left]; data[left++] = data[right]; data[right--] = temp;
    }
  }

unsigned int bintogray(n)
unsigned int n;
{
        int temp;

        temp = (n << 1);
        return (temp ^ n) >> 1;
}

unsigned int graytobin(g)
unsigned int g;
{
        int i, n;

        return (1);
}

void LandN (lSize, num, L, N)
int lSize[];
int num;
int *L, *N;
{
   int i, s;

   for (i = 1, s = 0; s <= num; i++)
      s += lSize[i];
```

```
   *L = i-1;
   *N = num - (s - lSize [i-1]);
}

#ifdef NODE
void send_synch(node)
int node;
{
   int i, flag;

   i = 1;
   nwrite (&i, sizeof (int), node, M_SYNCH, flag);
}
#endif

void get_synch(node)
int node;
{
   int from, type, flag, i;

   type = M_SYNCH;
   from = node;
#ifdef NODE
   nread (&i, sizeof (int), &from, &type, &flag);
#endif
#ifdef HOST
   printf ("Waiting for synchronization from %d...", from);
   nread (chan, &i, sizeof (int), &from, &type);
   printf ("Got it..\n");
#endif
}
#ifdef NODE
int dprintf ()
{
   int  flag;

   if (PRINTCOND)
      nwrite ((char *)dbuf, TEL_SIZE * sizeof (char), host, M_TELEGRAM, &flag);
}
#endif
```

```
/* pipehost.c

   Jeffrey Dean                                    CSci 5299
   ID 1321294                                      Term Project

   Back-propagation neural network - Host program
*/
#define HOST

#include <stdio.h>
#ifdef IBM
#include <stdlib.h>
#include <conio.h>
#endif
#include <math.h>
#include <ctype.h>
#include <string.h>
#include "pipe.h"

#ifdef IBM
int   ShowMatrix(MATRIX m, int nrows, int ncols);
#else
int   ShowMatrix();
#endif

/* define storage for net layers */
/* Arrays for inputs, outputs, deltas, weights & targets */
LAYER    net[MAXLAYERS];
int      lSize[MAXLAYERS];
VECTOR   PatternID;
MATRIX   target;      /* target output */
MATRIX   input;       /* input patterns */
char     *ibuf;

/* Cube communication variables */
int      chan;
int      cubedim, cubesize;
int      type;
char     *mbuf, *lbuf, *nflags;
int      b [SMALLBUFLEN];
float    bf[SMALLBUFLEN];
int      mSize;
float    avgtime;
float    tot_time;
int      time_cnt;

int      d1,d2;   /* variables used in delay variables */

#include "pipework.c"

#ifdef IBM
void  main(int argc, char *argv[])
#else
void  main(argc, argv)
int argc;
char *argv[];
#endif
{
   float eta    =  0.15,        /* default learning rate            */
         alpha  =  0.075;       /* default momentum factor          */
   int   nReportErrors = 1;     /* error reporting frequency        */
   float ErrorLevel = ERRORLEVEL; /* satisfactory error level       */
   float error;                 /* latest sum squared error value   */
   float neur_node;             /* patterns per node                */
   float neur_num;              /* current pattern number           */
   int   i;                     /* index variable                   */
   int   j;
   int   L, R;                  /* Left and right neighbors of nodes */
   int   startN, endN;          /* Starting and ending neurons      */
   int   l,                     /* layer index                      */
         n,                     /* neuron index                     */
         p,                     /* index pattern number             */
         nPatterns,             /* number of patterns desired       */
         nLayers,               /* number of layers in network      */
         nNeurons,              /* Number of hidden and output neurons */
         nIterations;           /* number of iterations desired     */
   FILE  *fpInp,                /* run file                         */
         *fpResults,            /* resulting weights file           */
         *fpWeights;            /* initial weight file              */
   char  szResults[66];         /* various filenames (pathnames)    */
   char  szWeights[66];
   char  *progname  =  *argv;   /* name of executable DOS 3.x only  */
   char  c;

   /* allocate message buffer */
   mbuf = (char *)malloc(MAXBUFLEN);
   ibuf = (char *)malloc(sizeof(INITIALMSG));

   printf ("Pipelined neural net simulation\n");

   /* read optional - arguments */
   for (; argc > 1;  argc--)
   {
      char *arg = *++argv;

      if (*arg  !=  '-')
         break;

      switch (*++arg)
      {
         case 'e':   sscanf(++arg,  "%d",  &nReportErrors);   break;
         case 'd':   sscanf(++arg,  "%f",  &ErrorLevel);      break;
         default:    break;
      }
   }

   if (argc < 2)
   {
      fprintf(stderr, "Usage:  %s [-en -df] runfilename\n",  progname);
      fprintf(stderr, "   -en   =>  report error every n iterations\n");
      fprintf(stderr, "   -df   =>  done if mean squared error < f\n");
      exit(1);
   }

   /* Open run file for reading */
   if ((fpInp = fopen(*argv, "r"))    ==   NULL)
   {
      fprintf(stderr, "%s: can't open file %s\n", progname, *argv);
      exit(1);
   }

   /* read and parse the run specification line; */
   fscanf(fpInp,
       "%s %s %d %d %f %f %d",
       szResults,          /* output results file */
       szWeights,          /* initial weights file */
       &nPatterns,         /* number of patterns to learn */
       &nIterations,       /* number of iterations through the data */
```

```
      &eta,              /* learning rate */
      &alpha,            /* momentum factor */
      &nLayers);         /* number of layers in network */

   /* get sizes of layers */
   nNeurons = 0;
   for (i = 0; i < nLayers; i++)
   {
      fscanf(fpInp, "%d", &lSize [i]);
      if (i != 0) nNeurons += lSize [i];
   }

   printf ("%d layers, %d patterns: ", nLayers, nPatterns);
   for (i = 0; i < nLayers; i++)
      printf (" %d", lSize [i]);
   printf ("\n");


   /*----------allocate dynamic storage for all data ---------------*/
   for (i = 0; i < nLayers; i++)
      LayerAllocate(&net [i], lSize [i], (i==0) ? 0:lSize [i-1]);

   MatrixAllocate(&target,    nPatterns,    lSize [nLayers-1]);
   MatrixAllocate(&input,     nPatterns,    lSize [0]);
   VectorAllocate(&PatternID, nPatterns);

   /*--------- Read the initial weight matrices: -------------------*/
   if ((fpWeights = fopen(szWeights,"r"))  ==  NULL)
   {
      fprintf(stderr,  "%s: can't open file %s\n",  progname, szWeights);
      exit(1);
   }

   /* read weights */
   for (l = 1; l < nLayers; l++)
      for (n = 0; n < net [l].lSize; n++)
         for (i = 0; i <= net [l-1].lSize; i++)
         {
            fscanf(fpWeights,  "%f",      &(net [l].w[n][i]));
            net [l].wdel[n][i] = 0.0;
         }

   fclose(fpWeights);

   /*------------ Read in all patterns to be learned -----------------*/
   for (p = 0;  p < nPatterns;  p++)
   {
      for (i = 0;   i < net [0].lSize;   i++)
         if (fscanf(fpInp,  "%f",   &input[p][i])  != 1)
            goto  ALLPATTERNSREAD;

      /* read in target outputs for input patterns read */
      for (i = 0;  i < net [nLayers-1].lSize;  i++)
         fscanf(fpInp,  "%f",   &target[p][i]);

      fscanf(fpInp, "%f", &PatternID[p]);
   }

ALLPATTERNSREAD:

   if (p < nPatterns)
   {
      fprintf(stderr, "%s:  %d out of %d patterns read\n",
              progname,  p,  nPatterns);
      nPatterns = p;
   }

/* All input data has been read.  Now start sending information to the nodes */

   cubedim = MAXCUBDIM + 1;
   while ((cubedim < 0) || (cubedim > MAXCUBDIM)) {
          printf(" Enter Cube Dimension (0 - %d)  : ", MAXCUBDIM);
          scanf("%d", &cubedim);
   }

   cubesize = power(2, cubedim);   /* number of processors */
   lbuf = (char *)malloc(LBUFLEN);
   printf ("Attempting to open cube: Try ");
   i = 0;
   do {
      printf ("%d..", i++);
   } while ((chan = nopen(cubedim)) < 0);
   printf ("Got channel %d\n", chan);

   /* load the node program */
   printf ("Loading node program.\n");
   nloadm(chan, "node.x", -cubesize, nflags, lbuf, LBUFLEN);
   free(lbuf);
   printf ("Node program loaded to %d processors.\n", cubesize);

   fprintf(stderr, "Setting up initial message to nodes.\n");
   ((IM_PTR)ibuf)->i_layers    = nLayers;
   ((IM_PTR)ibuf)->i_iters     = nIterations;
   ((IM_PTR)ibuf)->i_pats      = nPatterns;
   ((IM_PTR)ibuf)->i_seed      = 1L;
   ((IM_PTR)ibuf)->i_eta       = eta;
   ((IM_PTR)ibuf)->i_alpha     = alpha;
   ((IM_PTR)ibuf)->i_epsilon   = ErrorLevel;
   printf ("Set up initial message.  Now swapping.\n");
   swap (&((IM_PTR)ibuf)->i_layers, sizeof(int));
   swap (&((IM_PTR)ibuf)->i_iters, sizeof(int));
   swap (&((IM_PTR)ibuf)->i_pats, sizeof(int));
   swap (&((IM_PTR)ibuf)->i_seed, sizeof(long));
   swap (&((IM_PTR)ibuf)->i_eta, sizeof(float));
   swap (&((IM_PTR)ibuf)->i_alpha, sizeof(float));
   swap (&((IM_PTR)ibuf)->i_epsilon, sizeof(float));

   for (i = 0; i < MAXLAYERS; i++)
   {
      ((IM_PTR)ibuf)->i_lsize [i]  = lSize [i];
      swap (&((IM_PTR)ibuf)->i_lsize[i], sizeof(int));
   }

   neur_node = (float)nNeurons / (cubesize - 1)  /* -1 for splicer */;
   neur_num = 0;
   printf ("Now sending initial message to nodes.  Total neurons = %d\n",
            nNeurons);

   for (i = 0; i < cubesize; i++)
   {
      /* Compute left and right neighbors of node i */
      L = bintogray ((i==0) ? cubesize-1 : i-1);
      R = bintogray ((i + 1) % cubesize);
      ((IM_PTR)ibuf)->i_Lnbr = L;
      ((IM_PTR)ibuf)->i_Rnbr = R;
      /* Now calculate which neurons node i will process */
      if (i != 0)
      {
```

```
        startN  = (int) neur_num;
        endN    = (int) (neur_num + neur_node - 1);
        if (i == cubesize - 1)
           endN = nNeurons - 1;
        ((IM_PTR)ibuf)->i_startN  = startN;
        ((IM_PTR)ibuf)->i_endN    = endN;
        neur_num += neur_node;
     }
     else
     {
        startN = -1;
        endN = -1;
     }

     printf ("node %d (%d, %d) gets neurons %3d to %3d.\n", i,
     ((IM_PTR)ibuf)->i_Lnbr,
     ((IM_PTR)ibuf)->i_Rnbr,
     ((IM_PTR)ibuf)->i_startN,
     ((IM_PTR)ibuf)->i_endN);
     swap (&((IM_PTR)ibuf)->i_startN, sizeof(int));
     swap (&((IM_PTR)ibuf)->i_endN, sizeof(int));
     swap (&((IM_PTR)ibuf)->i_Lnbr, sizeof(int));
     swap (&((IM_PTR)ibuf)->i_Rnbr, sizeof(int));

     /* Send initial message to node */
     type = M_INITIAL;
     nwrite (chan, (char *)ibuf, sizeof(INITIALMSG), bintogray (i), type);
     if (i != 0) /* If not the splicer, send the weights */
        for (j = startN; j <= endN; j++)
        {
           printf ("Finding l and n for %d\n", j);
           LandN (lSize, j, &l, &n);
           printf ("Sending weights for %d,%d to %d.", l, n, bintogray(i));
           SendVector (net [l].w [n], lSize [l-1]+1, (VM_PTR)mbuf,
                       bintogray (i), M_WEIGHTS);
           for (d1=0; d1 < 100; d1++)
              for (d2=0; d2 < 1000; d2++)  /* nothing */ ;
           printf ("Done..\n");
        }
     else /* send the input and target output matrices to the splicer */
     {
        SendMatrix (input,  nPatterns, lSize [0], (MM_PTR)mbuf,
                    i, M_PATTERN);
        SendMatrix (target, nPatterns, lSize [nLayers-1], (MM_PTR)mbuf,
                    i, M_OUTPUT);
     }
     printf ("Waiting for synchronization from %d..", bintogray(i));
     get_synch (bintogray (i));
     printf ("Got it.\n");
  }

  printf ("Awaiting results..\n");

  /* Now we await the results */
  tot_time = 0;
  time_cnt = 0;
  j = 0;
  i = 0;
  do
  {  char c;
     long l;

     type = -1;
     while (ntest (chan, &i, &type) < 0)
     {
        i = (i + 1) % cubesize;
     }

     nread (chan, (char *)mbuf, TEL_SIZE, &i, &type);
     switch (type)
     {
        case M_TELEGRAM:
           printf ("Telegram  (%d): %s\n", i, (char *)mbuf);
           if (++j % 20 == 0)
           {
              printf ("--more--");
              scanf ("%c", &c);
           }
           break;
        case M_TIME:
           swap (mbuf, sizeof(int));
           printf ("TIME DATA (%d): %f ms (%d x %f)\n", i,
                    ((int *)mbuf)[0]*MS_PER_TICK,
                    ((int *)mbuf)[0], MS_PER_TICK);
           tot_time += ((int *)mbuf)[0] * MS_PER_TICK;
           time_cnt++;
           break;
        default:
           printf ("Non TELEGRAM from node %d", i);
     }
     i = (i + 1) % cubesize;
  } while (time_cnt < cubesize);

  printf ("Got past telegram receiving.\n");

  /* Now get the time taken from each node */
#ifdef DUMMY
  for (i = 0, avgtime = 0.0; i < cubesize; i++)
  {
     type = M_TIME;
     nread (chan, (char *)b, sizeof(int), &i, &type);
     swap (b, sizeof(int));
     printf ("Node %d took %f ms (%d x %f)\n", i, b[0]*MS_PER_TICK,
              b[0], MS_PER_TICK);
     avgtime += b[0] * MS_PER_TICK;
  }
#endif
  avgtime = tot_time / cubesize;
  printf ("\nAverage time per node: %6.1f ms\n", avgtime);

  printf ("Program terminating.\n");

  /*----------------- free dynamic storage for data ----------------------*/
  for (l = 0; l < nLayers; l++)
     LayerFree (&net [l]);
  MatrixFree(target,    nPatterns);
  MatrixFree(input,     nPatterns);
  free (PatternID);

  fclose(fpInp);                    /* close run file */
  nclose (chan);                    /* Close allocated hypercube */
}

#ifdef IBM
int   ShowMatrix(MATRIX m, int nrows, int ncols)
#else
int   ShowMatrix(m, nrows, ncols)
MATRIX m;
```

```
int nrows;
int ncols;
#endif
{
   int i, j;

   for (i = 0; i < nrows; i++)
   {
      for (j = 0; j < ncols; j++)
         printf ("%5.4f%c", m[i][j], (j==ncols-1) ? '\n' : ' ');
#ifdef IBM
      if (i % 25 == 24) getch();
#endif
   }
   return (1);
}
```

```
* pipenode.c

  Jeffrey Dean                                    CSci 5299
  ID 1321294                                      Term Project

  Pipelined back-propagation neural network
/
define NODE

ifdef IBM
include <stdlib.h>
endif
include <math.h>
include <ctype.h>
include <string.h>
include "pipe.h"

* define storage for net layers */
* Arrays for inputs, outputs, deltas, weights & targets */
nt      lSize[MAXLAYERS];
nt      nPatterns,              /* number of patterns desired          */
        nPats,                  /* Number of patterns in this message */
        nIterations,            /* number of iterations desired       */
        startN,                 /* starting neuron number for node    */
        endN,                   /* ending neuron number for node      */
        numN,                   /* number of neurons for this node    */
        nLayers,                /* number of layers in network        */
        maxLSize;               /* maximum layer size                 */
ECTOR   target;
ECTOR   in;
ECTOR   o;
ECTOR   del[MAXLAYERS];
EURON   N[MAXNEURONS];
* The following three matrices are only used on the splicer node */
ATRIX   targetM;
ATRIX   inputM;
ATRIX   deltaM;
nt      msg_stack;  /* number of messages that need to be added to pipeline */

nt      deltawidth;
nt      pat_msg_size;
nt      num_msgs;
nt      work_nodes;
loat    pat_per_msg;
nt      max_pats;
har     done;

nt      epoch;              /* epoch number */
loat    pptr;               /* next pattern to be loaded for splicer */
loat eta   =  0.15,         /* default learning rate            */
     alpha =  0.075;        /* default momentum factor          */
loat    error;              /* Sum of squares error for this epoch */
loat    lasterror;          /* Last error known to be completed */

ong     ranseed = 1235734L;

* Cube communication variables */
nt      nodeno, proc, host, cubedim, flag;
nt      type;
har     *mbuf;
nt      b [SMALLBUFLEN];
loat    bf[SMALLBUFLEN];
nt      left, right;        /* Left and right neighbors         */
nt      mtype;
```

```
#include "pipework.c"
void init_pointers();
void rotate_level();
void unroll();
void rollup();
void process_neuron();
void update_weights();
void setup_pat_msg();
void send_pat_msg();
void get_pat_msg();
void send_update_msg();
void get_update_msg();

void print_vec (s, v, sz)
char *s;
float v [];
int sz;
{
   int i;

   dbuf[0] = '\0';
   i = strlen(dbuf);
   sprintf(dbuf,"%s Len = %d %d %4.2f %4.2f %4.2f", s,i, sz, v[0],v[1],v[2]);
   dprintf();
   sprintf (dbuf, "%s (%d): ", s, sz);
   for (i = 0; i < sz; i++)
      sprintf (dbuf+strlen (dbuf), "%4.2f ", v [i]);
   dprintf();
}

void print_buffer(pNum)
int pNum;
{
   int i;

   init_pointers ((PAT_PTR)mbuf, pNum);
/*
   print_vec ("Input", in, lSize [0]);
   print_vec ("Target", target, lSize [nLayers-1]);
   print_vec ("Msg",&(((PAT_PTR)mbuf)->p_data[pNum*pat_msg_size]),pat_msg_size);
*/
}

void main()
{
   int   nReportErrors = 1;     /* error reporting frequency         */
   float ErrorLevel = ERRORLEVEL; /* satisfactory error level        */
   char  MonitorError = 0;      /* true when monitor error display   */
   int   i,                     /* index variables                   */
         j;
   int   n,                     /* neuron index                      */
         p,                     /* index pattern number              */
         q;                     /* index iteration number            */
   int   sTime;                 /* Starting time                     */

   whoami (&nodeno, &proc, &host, &cubedim);

   work_nodes = (1 << cubedim) - 1;

   /* allocate message buffer */
   mbuf = (char *)malloc(MAXBUFLEN);

   type = M_INITIAL;
```

```
read ((char *)mbuf, sizeof (INITIALMSG), &host, &type, &flag);

Layers     = ((IM_PTR)mbuf)->i_layers;
Iterations = ((IM_PTR)mbuf)->i_iters;
Patterns   = ((IM_PTR)mbuf)->i_pats;
tartN      = ((IM_PTR)mbuf)->i_startN;
ndN        = ((IM_PTR)mbuf)->i_endN;
eft        = ((IM_PTR)mbuf)->i_Lnbr;
ight       = ((IM_PTR)mbuf)->i_Rnbr;
anseed     = ((IM_PTR)mbuf)->i_seed;
ta         = ((IM_PTR)mbuf)->i_eta;
lpha       = ((IM_PTR)mbuf)->i_alpha;
rrorLevel  = ((IM_PTR)mbuf)->i_epsilon;

axLSize = 0;
or (i = 0; i < MAXLAYERS; i++)

   lSize [i] = ((IM_PTR)mbuf)->i_lsize [i];
   maxLSize = ((lSize [i] > maxLSize) ? lSize [i] : maxLSize);


* find number of neurons for this node */
umN = endN - startN + 1;

* Find width of all delta vectors */
eltawidth = 0;
or (i = 1; i < nLayers; i++)
   deltawidth += lSize [i];

* Now find width of pattern message */
at_msg_size = deltawidth + 2 * maxLSize + lSize [nLayers-1];

sprintf (dbuf, "Msg data: numN=%d, startN=%d,EndN=%d ni=%d np=%d %d %d %d %d",
umN, startN, endN,nIterations,nPatterns,nLayers,lSize[0],lSize[1],lSize[2]);
dprintf();

/* Now we have to find out how many patterns to send per message */
/* We can't have more than nPatterns / (# of work nodes) per message,
   or we won't be able to achieve full concurrency in our pipeline
   (the splicer will be forced to wait for the end of an epoch before
   it can start another epoch). */
num_msgs = (nPatterns + MAX_PAT_MSG - 1) / MAX_PAT_MSG;
sprintf (dbuf,"num_msgs = %d, MAX_PAT_MSG = %d, nPatterns=%d, wn=%d",num_msgs,
_PAT_MSG,nPatterns, work_nodes);
dprintf();
if (work_nodes > num_msgs)
   num_msgs = work_nodes;

/* patterns per message is # of patterns / number of messages */
pat_per_msg = (nPatterns + num_msgs - 1) / num_msgs;
if (pat_per_msg < 1)
   pat_per_msg = 1;

/* round to next highest integer */
max_pats = (int)pat_per_msg;
if (pat_per_msg != (float)max_pats)
   max_pats++;

epoch = 0;
done = FALSE;

/* get starting time */
sTime = ntime();
```

```
if (nodeno != 0)
{
   sprintf (dbuf,"Waiting for weight vectors from %d neurons.", numN);
   dprintf();
   /************************  Pipeline node section ****************/
   /* Allocate weight vectors for each neuron and get weights from host */
   for (n = 0; n < numN; n++)
   {
      int lev, neur;

      sprintf (dbuf, "Trying to get L and N for %d", n);
      dprintf();
      LandN (lSize, n + startN, &lev, &neur);
      N[n].nNum = neur;
      N[n].lNum = lev;
      sprintf (dbuf,"Waiting for %d (%d,%d), size = %d",
n, lev, neur, lSize [lev-1]+1);
      dprintf();
      VectorAllocate (N[n].w, lSize [lev-1]+1);
      VectorAllocate (N[n].wdel, lSize [lev-1]+1);
      for (i = 0; i <= lSize [lev-1]; i++)
         N[n].wdel [i] = 0;
      GetVector (N[n].w, lSize[lev-1]+1, (VM_PTR)mbuf, host, M_WEIGHTS);
   }
   send_synch (host);   /* Send sychronizing message to host */
   sprintf (dbuf, "Got weights for %d neurons, npats=%d (L,R) = %d,%d",
                  numN, nPatterns, left, right);
   dprintf();

   /* Main loop for pipeline nodes */
   while (!done)
   {
      mtype = get_message();
      switch (mtype)
      {
         case M_WUPDATE:
            get_update_msg();
            sprintf (dbuf, "Got update message %d of %d epochs done",
            ((UM_PTR)mbuf)->u_iter,((UM_PTR)mbuf)->u_maxiter);
            dprintf();
            send_update_msg(0.0);
            update_weights();
            if (((UM_PTR)mbuf)->u_iter >= ((UM_PTR)mbuf)->u_maxiter)
               done = TRUE;
            break;
         case M_PATTERN:
            get_pat_msg();
            for (p = 0; p < ((PAT_PTR)mbuf)->p_npats; p++)
            {
               init_pointers((PAT_PTR)mbuf, p);
               for (n = 0; n < numN; n++)
               {
                  process_neuron (n);
                  if (N[n].nNum == lSize [N[n].lNum] - 1)
                     rotate_level (N [n].lNum, p);
               }
            }
            send_pat_msg();
            break;
         default:
            sprintf (dbuf, "Unexpected message of type %d", mtype);
            dprintf();
            done = TRUE;
            break;
```

```
    }
  }
  /************************  End pipeline section *****************/
else

  /************************ splicer node section *****************/

  /* Allocate and get input and expected output matrices */
  MatrixAllocate (&inputM,  nPatterns, lSize [0]);
  MatrixAllocate (&targetM, nPatterns, lSize [nLayers-1]);
  GetMatrix (inputM,  nPatterns, lSize [0], (MM_PTR)mbuf, host, M_PATTERN);
  GetMatrix (targetM, nPatterns, lSize [nLayers-1], (MM_PTR)mbuf, host,
             M_OUTPUT);
  send_synch (host);   /* Synchronize with host */

  /* Allocate matrix to hold delta vectors */
  MatrixAllocate (&deltaM,  nPatterns, deltawidth);

  /* Initialize deltas to 0 */
  for (p = 0; p < nPatterns; p++)
     for (i = 0; i < deltawidth; i++)
        deltaM[p][i] = 0;

  sprintf (dbuf, "Splicer has been initialized.  deltawidth = %d",deltawidth);
  dprintf();

  pptr = 0;
  msg_stack = work_nodes;  /* set to fill pipeline initially */
  lasterror = 100.0;
  error = 0;

  while (!done)
  {
     while ((msg_stack > 0) && (!done))
     {
        sprintf (dbuf, "Sending patterns %f to %f (epoch %d)", pptr,
tr + pat_per_msg, epoch);
rintf();

tup_pat_msg (pptr, pptr + pat_per_msg);
        send_pat_msg();
        pptr += pat_per_msg;
        if (pptr >= nPatterns)
        {
           send_update_msg(lasterror);
           pptr = 0;
           if (epoch++ == nIterations)
              done = TRUE;
        }
        msg_stack--;
     }
     if (!done)
     {
     mtype = get_message();
     switch (mtype)
     {
        case M_WUPDATE:
           get_update_msg();
           sprintf (dbuf, "Got weight update back for epoch %d", epoch);
           dprintf();
           break;
        case M_PATTERN:
           get_pat_msg();
           rollup();
           /*
           sprintf (dbuf, "Got back and rolled up message for %d to %d",
               ((PAT_PTR)mbuf)->p_snum,
               ((PAT_PTR)mbuf)->p_snum+((PAT_PTR)mbuf)->p_npats);
           dprintf();
           */
           msg_stack++;  /* room for one more message in the pipeline */
           break;
        default:
           sprintf (dbuf, "Unexpected message of type %d", mtype);
           dprintf();
           done = TRUE;
           break;
     }
     }
  }  /* End of while */

  /************************ End splicer section *****************/
}

  /* Send time taken back to the host */
  b [0] = ntime() - sTime;
  nwrite ((char *)b, sizeof(int), host, M_TIME, &flag);

  endpcs();
}

int maxdiff (n1, n2)
int n1, n2;
{
  int i, j, mask;

  for (i = 0, j = 0; i < 8 * sizeof (int); i++)
  {
     mask = (1 << i);
     if ((n1 & mask) != (n2 & mask))
        j = i;
  }
  if (n1 == n2)
     return (-1);
  else
     return (j);
}

void init_pointers(pbuf, pNum)
PAT_PTR pbuf;
int     pNum;
/* Init_pointers calculates the offsets into the pattern buffer for
   the target vector, input and output vectors, and the delta vectors.
   The data is arranged as follows:

      Offset     Data
           0:    Target vector
     S[nL-1]:    Delta vector 1
       +S[1]:    Delta vector 2
                 ...
    +S[nL-2]:    Delta vector nL-1
    +S[nL-1]:    Input vector
   +maxLSize:    Output vector
*/
{
  int i, temp, t2;

  /* Calculate offset of pth pattern into message buffer */
```

```
  t2 = pat_msg_size * pNum;

  /* Now calculate offsets into pattern data array */
  target = &(pbuf->p_data[0+t2]);
  temp = lSize [nLayers-1];   /* Target is output vector for last layer */
  for (i = 1; i < nLayers; i++)
  {
     del[i] = &(pbuf->p_data[temp+t2]);
     temp += lSize [i];
  }
  in = &(pbuf->p_data[temp+t2]);
  o = &(pbuf->p_data[temp+maxLSize+t2]);


oid rotate_level(from_lev, pNum)
nt     from_lev;
nt     pNum;

  int i;

  if (from_lev != nLayers - 1)
  {
     /* Initialize pointers for this pattern */
     init_pointers ((PAT_PTR)mbuf, pNum);

     /* First step: Copy output vector to input vector */
     for (i = 0; i < maxLSize; i++)
        in [i] = o [i];

     /* Now zero the delta vector for the level we're coming from */
     for (i = 0; i < lSize [from_lev]; i++)
        del[from_lev][i] = 0;
  }


oid process_neuron (NN)
nt     NN;

  int i, l, n;
  float sum, dw;

  /* Get nNum and lNum into n and l for convenience */
  n = N[NN].nNum;
  l = N[NN].lNum;

**** Step I.  Calculate new activation values given the input vector */
  /* Initialize with bias */
  sum = N[NN].w[lSize[l-1]];

  for (i = 0; i < lSize [l-1]; i++)
     sum += in [i] * N[NN].w [i];
  o [n] = f (sum);

**** Step II. Calculate deltas */
  /* If we're an output layer, we have to first calculate the error from
     the expected output */
  if (l == nLayers - 1)
     del [l][n] = (target [n] - o [n]);

  /* Now scale the sum of the errors by the derivative of the f function */
  del [l][n] *= o [n] * (1.0 - o [n]);

  if (l != 1)  /* if not the first hidden layer */
  {
     /* Now back-propogate using the delta values from the current layer */
     for (i = 0; i < lSize [l - 1]; i++)
        del [l-1][i] += N[NN].w [i] * del [l][n];
  }
/**** Step III. Calculate weight changes based on deltas we just calculated */
  dw = eta * del [l][n];
  N[NN].wdel [lSize [l-1]] += dw;

  for (i = 0; i < lSize [l-1]; i++)
  {
     dw = eta * del [l][n] * in [i];
           /* input to this layer is output from the previous layer */
     N[NN].wdel [i] += dw;
  }
}

void unroll (p, pnum)
int     p;     /* slot number to put pattern in */
int     pnum;  /* pattern to be unrolled */
{
  int i;

  /* First we have to initialize the pointers to the correct locations */
  init_pointers ((PAT_PTR)mbuf, p);
  /* Copy target pattern to target vector */
  for (i = 0; i < lSize (nLayers-1]; i++)
     target [i] = targetM[p][i];
  /* Copy delta vectors.  Note that deltas are contiguous in the message
     buffer, so we can just copy it as if it was one vector. */
  for (i = 0; i < deltawidth; i++)
     del[1][i] = deltaM[p][i];
  /* Copy input pattern to activation vector and zero rest of activation
     vector and all of the output vector. */
  for (i = 0; i < maxLSize; i++)
  {
     in [i] = (i < lSize [0] ? inputM [p][i] : 0);
     o  [i] = 0;
  }
}

void rollup ()
{
  int i, p, npats, sp;
  float temp;                    /* Temporary value to hold error term */

  npats = ((PAT_PTR)mbuf)->p_npats;
  sp = ((PAT_PTR)mbuf)->p_snum;
  for (p = 0; p < npats; p++)
  {
     init_pointers ((PAT_PTR)mbuf, p);
     for (i = 0; i < deltawidth; i++)
        deltaM [p+sp][i] = del [1][i];
     for (i = 0; i < lSize [nLayers-1]; i++)
     {
        temp = target [i] - o [i];
        error += temp * temp;
     }
     if (sp + p == nPatterns - 1)  /* If last pattern in the epoch.. */
     {
        lasterror = error / (nPatterns * lSize [nLayers-1]);
        sprintf (dbuf, "END of EPOCH %d: Error = %f", epoch, lasterror);
        dprintf();
        error = 0;  /* Initialize error for next epoch */
     }
```

```
}

id setup_pat_msg (start_pat, end_pat)
oat   start_pat;
oat   end_pat;

 int p, npats;

 npats = 0;
 ((PAT_PTR)mbuf)->p_snum = (int)start_pat;
 for (p = (int)start_pat; p < end_pat; p++)
 if (p < nPatterns)
    {
       unroll (p, npats);
       npats++;
    }
 ((PAT_PTR)mbuf)->p_npats = npats;


id send_pat_msg()

 int msgsize;

 msgsize = pat_msg_size * max_pats * sizeof (float) + 2 * sizeof (int);
 nwrite ((char *)mbuf, msgsize, right, M_PATTERN, &flag);


id get_pat_msg()

 int msgsize, mtype;

 mtype = M_PATTERN;
 msgsize = pat_msg_size * max_pats * sizeof (float) + 2 * sizeof (int);
 nread ((char *)mbuf, msgsize, &left, &mtype, &flag);


id send_update_msg(e)
oat   e;

 if (nodeno == 0)  /* If we're the splicer, set up the message */
 {
    ((UM_PTR)mbuf)->u_iter = epoch;
    ((UM_PTR)mbuf)->u_maxiter = nIterations;
    ((UM_PTR)mbuf)->u_error = e;
    ((UM_PTR)mbuf)->u_error = e;
 }
 nwrite ((char *)mbuf, sizeof (UPDATEMSG), right, M_WUPDATE, &flag);


id get_update_msg()

 int mtype;

 mtype = M_WUPDATE;
 nread ((char *)mbuf, sizeof (UPDATEMSG), &left, &mtype, &flag);


t get_message()

 int result, mtype;

 do {
    mtype = -1;
    result = ntest (&left, &mtype);
 } while (result < 0);
 return (mtype);
}

void update_weights()
{
   int i, n;

   for (n = 0; n < numN; n++)
   {  /* sprintf (dbuf, "Weight delta for %d: ", n+startN); */
      for (i = 0; i <= lSize [N[n].lNum - 1]; i++)
      {
         N[n].w[i] += N[n].wdel[i];
         /* sprintf (dbuf+strlen(dbuf), "%f ", N[n].wdel[i]); */
         N[n].wdel[i] *= 0;  /* keep momentum */
      }
   /* dprintf(); */
   }
}
```